(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
16 October 2003 (16.10.2003)

PCT

(10) International Publication Number
WO 03/085517 A1

(51) International Patent Classification[7]: G06F 9/30, 9/38

(21) International Application Number: PCT/US03/09478

(22) International Filing Date: 27 March 2003 (27.03.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
10/116,670 3 April 2002 (03.04.2002) US

(71) Applicant: INTEL CORPORATION (A DELAWARE CORPORATION) [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95052 (US).

(72) Inventors: WOLRICH, Gilbert; 4 Cider Mill Road, Framingham, MA 01701 (US). ROSENBLUTH, Mark; 4 Crestview Drive, Uxbridge, MA 01569 (US). BERNSTEIN, Debra; 443 Peakham Road, Sudbury, MA 01776 (US). ADILETTA, Matthew; 20 Monticello Drive, Worcester, MA 01603 (US).

(74) Agents: HARRIS, Scott, C. et al.; Fish & Richardson, P.C., 4350 La Jolla Village Drive, Suite 500, San Diego, CA 92122 (US).

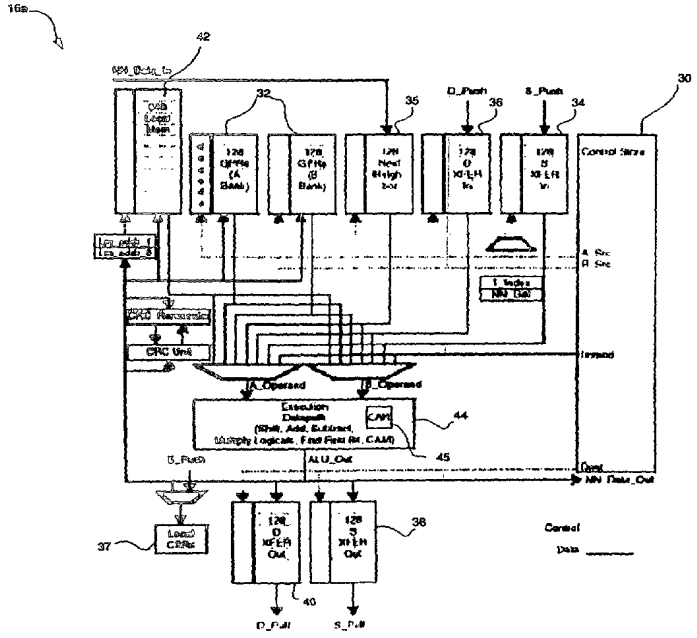
(81) Designated States (national): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States (regional): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, RO, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— with international search report

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: REGISTERS FOR DATA TRANSFERS WITHIN A MULTITHREADED PROCESSOR



(57) Abstract: A system and method for employing registers for data transfer in multiple hardware contexts and programming engines to facilitate high performance data processing. The system and method includes a processor that includes programming engines with registers for transferring data from one of the registers residing in an executing programming engine to a subsequent one of the registers residing in an adjacent programming engine.

WO 03/085517 A1

REGISTERS FOR DATA TRANSFERS WITHIN A MULTITHREADED PROCESSOR

BACKGROUND

Parallel processing is an efficient form of information processing of concurrent events in a computing process. Parallel processing demands concurrent execution of many programs, in contrast to sequential processing. In the context of parallel processing, parallelism involves doing more than one thing at the same time. Unlike a serial paradigm where all tasks are performed sequentially at a single station or a pipelined machine where tasks are performed at specialized stations, with parallel processing, many stations are provided, each capable of performing and carrying out various tasks and functions simultaneously. A number of stations work simultaneously and independently on the same or common elements of a computing task. Accordingly, parallel processing solves various types of computing tasks and certain problems are suitable for solution by applying several instruction processing units and several data streams.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a processing system.

FIG. 2 is a detailed block diagram of the processing system of FIG. 1 where one of the embodiments of the invention may be advantageously practiced.

FIG. 3 is a block diagram of a functional pipeline unit of the processing system of FIG. 1.

FIG. 4 is a block diagram illustrating details of the processing system of FIG. 1 where one of the embodiments of the invention may be advantageously practiced.

FIG. 5 is a simplified block diagram of a context pipeline process.

FIG. 6 is a flowchart illustrating the process of a context pipeline where one of the embodiments of the invention may be advantageously practiced.

FIG. 7 is a flowchart illustrating the process of determining the address of the Next Neighbor registers.

DESCRIPTION

Architecture:

Referring to FIG. 1, a computer processing system 10 includes a parallel, hardware-based multithreaded network processor 12. The hardware-based multithreaded processor 12 is coupled to a memory system or memory resource 14. Memory system 14 includes dynamic random access memory (DRAM) 14a and static random access memory 14b (SRAM). The processing system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. Specifically, the hardware-based multithreaded processor 12 is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has

multiple functional microengines or programming engines 16a-16h (collectively, programming engines 16) each with multiple hardware controlled threads that are simultaneously active and independently work on a specific task.

The programming engines 16 each maintain program counters in hardware and states associated with the program counters. Effectively, corresponding sets of context or threads can be simultaneously active on each of the programming engines 16 while only one is actually operating at any one time.

In this example, eight programming engines 16a-16h are illustrated in FIG. 1. Each programming engine 16a-16h processes eight hardware threads or contexts. The eight programming engines 16a-16h operate with shared resources including memory resource 14 and bus interfaces (not shown). The hardware-based multithreaded processor 12 includes a dynamic random access memory (DRAM) controller 18a and a static random access memory (SRAM) controller 18b. The DRAM memory 14a and DRAM controller 18a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SRAM memory 14b and SRAM controller 18b are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the core processor 20, and the like.

The eight programming engines 16a-16h access either the

DRAM memory 14a or SRAM memory 14b based on characteristics of the data. Thus, low latency, low bandwidth data are stored in and fetched from SRAM memory 14b, whereas higher bandwidth data for which latency is not as important, are stored in and fetched from DRAM memory 14a. The programming engines 16 can execute memory reference instructions to either the DRAM controller 18a or SRAM controller 18b.

The hardware-based multithreaded processor 12 also includes a processor core 20 for loading microcode control for the programming engines 16. In this example, although other types of processor cores may be used in embodiments of this invention, the processor core 20 is an XScale™ based architecture, designed by Intel® Corporation, of Santa Clara, CA.

The processor core 20 performs general-purpose computer type functions such as handling protocols, exceptions, and extra support for packet processing where the programming engines 16 pass the packets off for more detailed processing such as in boundary conditions.

The processor core 20 executes an operating system (not shown). Through the operating system (OS), the processor core 20 can call functions to operate on the programming engines 16a-16h. For the core processor 20 implemented as an XScale™ architecture, operating systems such as Microsoft® NT real-time of Microsoft® Corporation, of Seattle, Washington, VxWorks® real-time operating system of

WindRiver®, of Alameda, California, or a freeware OS available over the Internet can be used.

Advantages of hardware multithreading can be explained by SRAM or DRAM memory accesses. As an example, an SRAM access requested by a context (e.g., Thread_0), from one of the programming engines 16, e.g., programming engine 16a, will cause the SRAM controller 18b to initiate an access to the SRAM memory 14b. The SRAM controller 18b accesses the SRAM memory 14b, fetches the data from the SRAM memory 14b, and returns data to a requesting programming engine 16.

During an SRAM access, if one of the programming engines 16a-16h has a single thread that could operate, that programming engine would be dormant until data was returned from the SRAM memory 14b.

By employing hardware context swapping within each of the programming engines 16a-16h, the hardware context swapping enables other contexts with unique program counters to execute in that same programming engine. Thus, another thread e.g., Thread_1 can function while the first thread, Thread_0, is awaiting the read data to return. During execution, Thread_1 may access the DRAM memory 14a. While Thread_1 operates on the DRAM unit, and Thread_0 is operating on the SRAM unit, a new thread, e.g., Thread_2 can now operate in the programming engine 16. Thread_2 can operate for a certain amount of time until it needs to access memory or perform some other long latency operation,

such as making an access to a bus interface. Therefore, simultaneously, the multi-threaded processor 12 can have a bus operation, an SRAM operation, and a DRAM operation all being completed or operated upon by one of the programming engines 16 and have one more threads or contexts available to process more work.

The hardware context swapping also synchronizes the completion of tasks. For example, two threads can access the shared memory resource, e.g., the SRAM memory 14b. Each one of the separate functional units, e.g., the SRAM controller 18b, and the DRAM controller 18a, when they complete a requested task from one of the programming engine threads or contexts reports back a flag signaling completion of an operation. When the programming engines 16a-16h receive the flag, the programming engines 16a-16h can determine which thread to turn on.

One example of an application for the hardware-based multithreaded processor 12 is as a network processor. As a network processor, the hardware-based multithreaded processor 12 interfaces to network devices such as a Media Access Controller (MAC) device, e.g., a 10/100BaseT Octal MAC or a Gigabit Ethernet device compliant with IEEE 802.3. In general, as a network processor, the hardware-based multithreaded processor 12 can interface to any type of communication device or interface that receives or sends large amount of data. The computer processing system 10

functioning in a networking application can receive network packets and process those packets in a parallel manner.

Registers in Programming Engines:

Referring to FIG. 2, one exemplary programming engine 16a from the programming engines 16, is shown. The programming engine 16a includes a control store 30, which in one example includes a RAM of 4096 instructions, each of which is 40-bits wide. The RAM stores a microprogram that the programming engine 16a executes. The microprogram in the control store 30 is loadable by the processor core 20 (FIG. 1).

In addition to event signals that are local to an executing thread, the programming engine 16a employs signaling states that are global. With signaling states, an executing thread can broadcast a signal state to all programming engines 16a-16h. Any and all threads in the programming engines can branch on these signaling states.

As described above, the programming engine 16a supports multi-threaded execution of eight contexts. This allows one thread to start executing just after another thread issues a memory reference and must wait until that reference completes before doing more work. Multi-threaded execution is critical to maintaining efficient hardware execution of the programming engine 16a because memory latency is significant. Multi-threaded execution allows the programming engines 16 to hide memory latency by performing useful

independent work across several threads.

The programming engine 16a, to allow for efficient context swapping, has its own register set, program counter, and context specific local registers. Having a copy per context eliminates the need to move context specific information to and from shared memory and programming engine registers for each context swap. Fast context swapping allows a context to perform computations while other contexts wait for input-output (I/O), typically, external memory accesses to complete or for a signal from another context or hardware unit.

General Purpose Registers

The programming engine 16a executes the eight contexts by maintaining eight program counters and eight context relative sets of registers. A number of different types of context relative registers, such as general purpose registers (GPRs) 32, inter-programming agent registers (not shown), Static Random Access Memory (SRAM) input transfer registers 34, Dynamic Random Access Memory (DRAM) input transfer registers 36, SRAM output transfer registers 38, DRAM output transfer registers 40.

The GPRs 32 are used for general programming purposes. The GPRs 32 are read and written exclusively under program control. The GPRs 32, when used as a source in an instruction, supply operands to an execution datapath 44.

The execution datapath 44 can take one or two operands, perform an operation, and optionally write back a result. The execution datapath 44 includes a content addressable memory (CAM) 45. Each entry of the CAM 45 stores a 32-bit value, which can be compared against a source operand. All entries are compared in parallel and the result of the lookup is a 6-bit value.

When used as a destination in an instruction, the GPRs 32 are written with the result of the execution datapath 44. The programming engine 16a also includes I/O transfer registers 34, 36, 38 and 40 which are used for transferring data to and from the programming engine 16a and locations external to the programming engines 16a, e.g., the DRAM memory 14a, the SRAM memory 14b, and the like.

Transfer Registers

The programming engine 16a also includes transfer registers 34, 36, 38 and 40. Transfer registers 34, 36, 38 and 40 are used for transferring data to and from the programming engine 16a and locations external to the programming engine, e.g., DRAMs, SRAMs etc. There are four types of transfer registers as illustrated in FIG. 2, namely, input transfer registers and output transfer registers.

The input transfer registers, when used as a source in an instruction, supply operands to the execution datapath

44, whereas output transfer registers are written with the result from the execution datapath 44 when utilized as a destination in an instruction.

Local Control and Status Registers (CSRs)

Local control and status registers (CSRs) 37 are external to the execution datapath 44 and hold specific purpose information. They can be read and written by special instructions (local_csr_ rd and local_csr_wr) and are typically accessed less frequently than datapath registers.

Next Neighbor Registers

The programming engine 16a also includes one hundred and twenty eight (128) Next Neighbor (NN) registers, collectively referred to as NN registers 35. Each NN Register 35, when used as a source in an instruction, also supplies operands to the execution datapath 44. Each NN register 35 is written either by an external entity, not limited to, an adjacent programming engine, or by the same programming engine 16a where each NN register 35 resides. The specific register is selected by a context-relative operation where the register number is encoded in the instruction, or as a ring operation, selected via, e.g., NN_Put (NN write address) and NN_Get (NN read address) in the CSR Registers.

NN_Put registers are used when the previous neighboring

programming engine executes an instruction with NN_Put as a destination. The NN register selected by the value in this register is written, and the value in NN_Put is then incremented (a value of 127 wraps back to 0). The value in this register is compared to the value in NN_Get register to determine when to assert NN_Full and NN_Empty status signals.

NN_Get registers are used when the NN register 35 is accessed as a source, which is specified in the source field of the instruction. The NN register selected by the value in this register is read, and the value in NN_Put is then decremented (a value of 127 wraps back to 0). The value in this register is compared to the value in the NN_Put register to determine when to assert NN_Full and NN_Empty status signals.

Specifically, when each NN register 35 is used as a destination in an instruction, the instruction result data are sent out of the programming engine 16a, typically to another, adjacent programming engine. On the other hand, when the NN register 35 is used as a destination in an instruction, the instruction result data are written to the selected NN Register 35 in the programming engine 16a. The data are not sent out of the programming engine 16a as it would be when each NN register 35 is used as a destination. Each NN register 35 is used in a context pipelining method, as described below.

A local memory 42 is also used. The local memory 42 includes addressable storage located in the programming engine 16a. The local memory 42 is read and written exclusively under program control. The local memory 42 also includes variables shared by all the programming engines 16. Shared variables are modified in various assigned tasks during functional pipeline stages by the programming engines 16a-16h, which are described next. The shared variables include a critical section, defining the read-modify-write times. The implementation and use of the critical section in the computing processing system 10 is also described below.

Functional Pipelining and Pipeline Stages

Referring to FIG. 3, the programming engine 16a is shown in a functional pipeline unit 50. The functional pipeline unit 50 includes the programming engine 16a and a data unit 52 that includes data, operated on by the programming engine, e.g., network packets 54. The programming engine 16a is shown having a local register unit 56. The local register unit 56 stores information from the data packets 54.

In the functional pipeline unit 50, the contexts 58 of the programming engines 16a, namely, Programming Engine0.1 (PE0.1) through Programming Engine0.n (PE0.n), remain with the programming engine 16a while different functions are

performed on the data packets 54 as time 66 progresses from time = 0 to time = t. A programming execution time is divided into "m" functional pipeline stages or pipe-stages 60a-60m. Each pipeline stage of the pipeline stages 60a-60m performs different pipeline functions 62a, 64, or 62p on data in the pipeline.

The pipeline stage 60a is, for example, a regular time interval within which a particular processing function, e.g., the function 62a is applied to one of the data packets 54. A processing function 62 can last one or more pipelines stages 60. The function 64, for example, lasts two pipeline stages, namely pipeline stages 60b and 60c.

A single programming engine such as the programming engine 16a can constitute a functional pipeline unit 50. In the functional pipeline unit 50, the functions 62a, 64, and 62p move through the functional pipeline unit 50 from one programming engine (e.g., programming engine 16a), to another programming engine (e.g., programming engine 16b), as will be described next.

Referring to FIG. 4, the data packets 54 are assigned to programming engine contexts 58 in order. Thus, if "n" threads or contexts 58 execute in the programming engine 16a, the first context 58, "PE0.1" completes processing of the data packet 54 before the data packets 54 from the "PE0.n" context arrives. With this approach the programming engine 16b can begin processing the "n+1" packet.

Dividing the execution time of the programming engine 16a, for example, into functional pipeline stages 60a-60c results in more than one of the programming engines 16 executing an equivalent functional pipeline unit 70 in parallel. The functional pipeline stage 60a is distributed across two programming engines 16a and 16b, with each of the programming engines 16a and 16b executing eight contexts each.

In operation, each of the data packets 54 remains with one of the contexts 58 for a longer period of time as more programming engines 16 are added to the functional pipeline units 50 and 70. In this example, the data packet 54 remains with a context sixteen data packet arrival times (8 contexts x 2 programming engines) because context PE0.1 is not required to accept another data packet 58 until the other contexts 58 have received their data packets.

In this example, function 62a of the functional pipeline stage 60a can be passed from the programming engine 16a to the programming engine 16b. Passing of the function 62a is accomplished by using Next Neighbor registers, as illustrated by dotted lines 80a-80c in FIG. 4.

The number of functional pipeline stages 60a-60m is equal to the number of the programming engines 16a and 16b in the functional pipeline units 50 and 70. This ensures that a particular pipeline stage executes in only one programming engine 16 at any one time.

Context Pipelining:

Each of the programming engine 16 supports multi-threaded execution of eight contexts. One reason for this is to allow one thread to start executing just after another thread issues a memory reference and must wait until that reference completes before doing more work. This behavior is critical to maintaining efficient hardware execution of the programming engines 16a-16f because memory latency is significant. Stated differently, if only a single thread execution was supported, the programming engine would sit idle for a significant number of cycles waiting for references to complete and thereby reduce overall computational throughput. Multi-threaded execution allows a programming engine to hide memory latency by performing useful independent work across several threads.

The programming engines 16a-16h (Fig. 1) each have eight available contexts. To allow for efficient context swapping, each of the eight contexts in the programming engine has its own register set, program counter, and context specific local registers. Having a copy per context eliminates the need to move context specific information to/from shared memory and programming engine registers for each context swap. Fast context swapping allows a context to do computation while other contexts wait for I/O, typically external memory accesses, to complete or for a signal from

another context or hardware unit.

Referring now to FIG. 5, the context for a specific assigned task is maintained on the programming engines 16a-16c using CAM 45a-45c. The packets are processed in a pipelined fashion similar to an assembly line using NN registers 35a-35c to pass data from one programming engine to a subsequent, adjacent programming engine. Data are passed from one stage 90a to a subsequent stage 90b and then from stage 90b to stage 90c of the pipeline, and so forth. In other words, data are passed to the next stage of the pipeline allowing the steps in the processor cycle to overlap. In particular, while one instruction is being executed, the next instruction can be fetched, which means that more than one instruction can be in the "pipe" at any one time, each at a different stage of being processed.

For example, data can be passed forward from one programming engine 16 to the next programming engine 16 in the pipeline using the NN registers 35a-35c, as illustrated by example in FIG. 5. This method of implementing pipelined processing has the advantage that the information included in CAM 45a-45c for each stage 90a-c is consistently valid for all eight contexts of the pipeline stage. The context pipeline method may be utilized when minimal data from the packet being processed must advance through the context pipeline.

Referring to FIG. 6, as described above, context

pipelining requires that the data resulting from a pipe stage, such as pipe stage P, be sent to the next pipe stage, e.g., pipe stage P+1 (100). Then, Next Neighbor registers can be written from the ALU output of the processing engine 16a in pipe stage P (102), and the Next Neighbor registers can be read as a source operand by the next programming engine 16b at the pipe stage P+1 (104).

Referring to FIG. 7, two processes may be used to determine the address of the Next Neighbor registers to be written in the programming engine 16b. In one process, each context of the programming engine 16a may write to the same Next Neighbor registers for the same context in programming engine 16b (200). In another method, a write pointer register in the programming engine 16a and a read pointer register in the programming engine 16b may be used (300) to implement an inter processing engine FIFO (302). The values of write pointer register in the programming engine 16a and the read pointer register in the programming engine 16b are used to produce a full indication checked by the programming engine 16 before inserting data onto the FIFO (304), and an empty indication may be used the programming engine 16b before removing data from the FIFO (306). The FIFO Next Neighbor configuration may provide the elasticity between contexts in the pipe stages P and P+1. When a context in the pipe stage P+1 finds the Next Neighbor FIFO is empty, that context can perform a No-op function, allowing the pipe

stage to maintain a predetermined execution rate or "beat" even if the previous pipe stage may not be supplying an input at this same rate.

Other Embodiments:

In the examples described above in conjunction with FIGS. 1-7, the computer processing system 10 may implement programming engines 16 using a variety of network processors.

It is to be understood that while the invention has been described in conjunction with the detailed description thereof, the foregoing description is intended to illustrate and not limit the scope of the invention, which is defined by the scope of the appended claims. Other aspects, advantages, and modifications are within the scope of the following claims.

WHAT IS CLAIMED IS:

1. A processor comprising:

a plurality of programming engines that includes a plurality of registers for transferring data from one of the plurality of registers residing in an executing programming engine to a subsequent one of the plurality of registers residing in an adjacent programming engine.

2. The processor of claim 1 wherein the plurality of registers comprises next neighbor registers.

3. The processor of claim 1 wherein the plurality of registers are configured to assign tasks for packet processing to the plurality of programming engines.

4. The processor of claim 1 wherein the plurality of registers are configured to establish programming stages corresponding to the plurality of programming engines.

5. The processor of claim 1 wherein the plurality of registers establish a plurality of pipelines between the programming stages.

6. The processor of claim 1 wherein the plurality of registers maintain the currently operating programming stage of the pipeline and the adjacent programming engine is

configured to maintain a subsequent programming stage of the plurality of pipelines.

7. The processor of claim 1 wherein the plurality of registers support a functional pipeline by a functional pipeline control unit that passes functional data among the plurality of programming engines.

8. The processor of claim 7 further comprising a synchronization unit across the functional pipeline unit.

9. The processor of claim 7 wherein the functional pipeline control unit includes a plurality of functional pipeline stages.

10. The processor of claim 7 wherein each of the plurality of functional pipeline stages perform a different system function.

11. The processor of claim 7 wherein the plurality of programming engines process a data packet in order.

12. The processor of claim 7 wherein the data packet are assigned to multiple contexts of the plurality of programming engines.

13. The processor of claim 7 wherein the plurality of programming engines execute a data packet processing function using the functional pipeline unit of the system.

14. The processor of claim 7 wherein the plurality of programming engines perform inter-thread signaling.

15. A method of transferring data between programming engines, the method comprising:

reading data from a plurality of data registers of the programming engines for processing the data in a parallel processor of a pipeline unit, which supports execution of multiple contexts in each of the programming engines; and

writing data to the plurality of data registers of the programming engines, the plurality of data registers residing in an executing programming engine.

16. The method of claim 15 further comprising providing next neighbor registers for reading data from the plurality of data registers and writing data to the plurality of data registers.

17. The method of claim 15 further comprising assigning tasks for packet processing to the plurality of programming engines.

18. The method of claim 15 further comprising establishing programming stages corresponding to the plurality of programming engines and to establish a plurality of pipelines between the programming stages.

19. The method of claim 15 further comprising maintaining the currently operating programming stage of the pipeline and configuring the adjacent programming engine to maintain a subsequent programming stage of the plurality of pipelines.

20. The method of claim 15 wherein the reading and the writing of data includes supporting a functional pipeline by a functional pipeline control unit that passes functional data among the plurality of programming engines.

21. A computer program product stored on a computer readable medium, the program comprising instructions for causing a parallel processor to:

read data from a plurality of data registers of the programming engines for processing data in a parallel processor of a pipeline unit, which supports execution of multiple contexts in each of the programming engines; and

write data to the plurality of data registers of the programming engines, the plurality of data registers residing in an executing programming engine.

22. The computer program product of claim 21 further comprising instructions causing the processor to provide next neighbor registers for reading data from the plurality of data registers and writing data to the plurality of data registers.

23. The computer program product of claim 21 wherein the plurality of registers execute data transfers using the pipeline unit.

24. The computer program product of claim 21 further comprising instructions causing the processor to configure the plurality of data registers to assign tasks for packet processing to the plurality of programming engines.

25. An article comprising:

a storage medium having stored thereon instructions that when executed by a machine results in the following: transfer of data from one of a first plurality of registers in an executing programming engine to a subsequent one of a second plurality of registers residing in an adjacent programming engine;

assigning of tasks for packet processing to the plurality of programming engines;

establishing programming stages corresponding to the plurality of programming engines; and

establishing a plurality of pipelines between the programming stages.

26. The computer program product of claim 25 further comprising instructions to transfer data from a next neighbor register residing in a currently executing programming engine of the programming engines to a subsequent next neighbor register residing in programming engine adjacent to the currently executing programming engine.

27. The computer program product of claim 25 further comprising instructions for the plurality of registers to maintain the currently operating programming stage of the pipeline and for the adjacent programming engine to maintain a subsequent programming stage of the plurality of pipelines.

28. A multiprocessing system comprising:

a plurality of programming engines that includes a plurality of next neighbor registers configured to transfer data from one of a first plurality of registers residing in an executing programming engine to a subsequent one of a second plurality of registers residing in an adjacent

programming engine and a synchronization unit to process data packets across a functional pipeline unit.

29. The multiprocessing system of claim 28 further comprising shared memory locations utilized by a plurality of programming stages of the plurality of programming engines, the shared variables including a critical section defining the read-modify-write time of the memory locations.

30. The multiprocessing system of claim 28 wherein each of the plurality of programming engines further includes a content addressable memory (CAM).

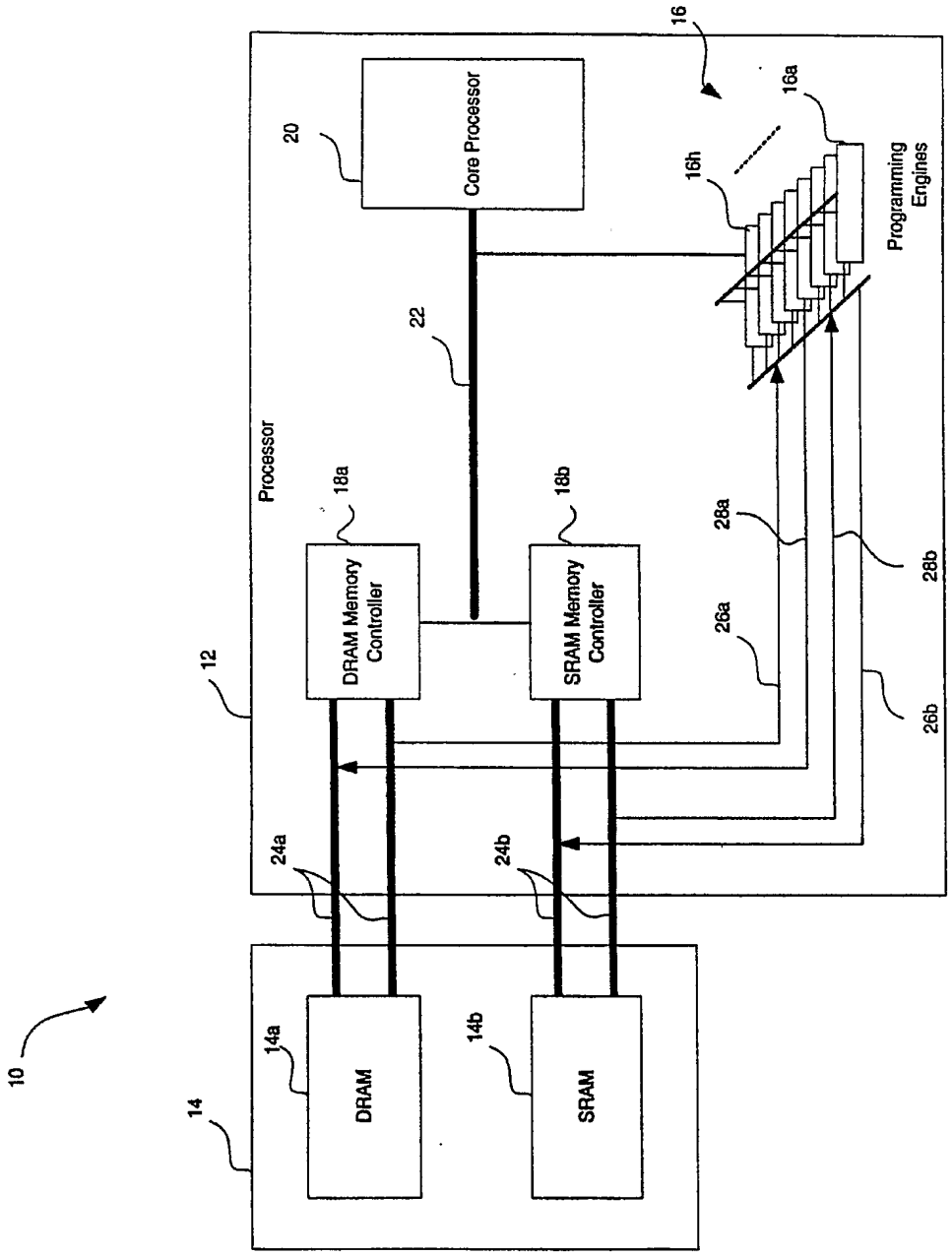
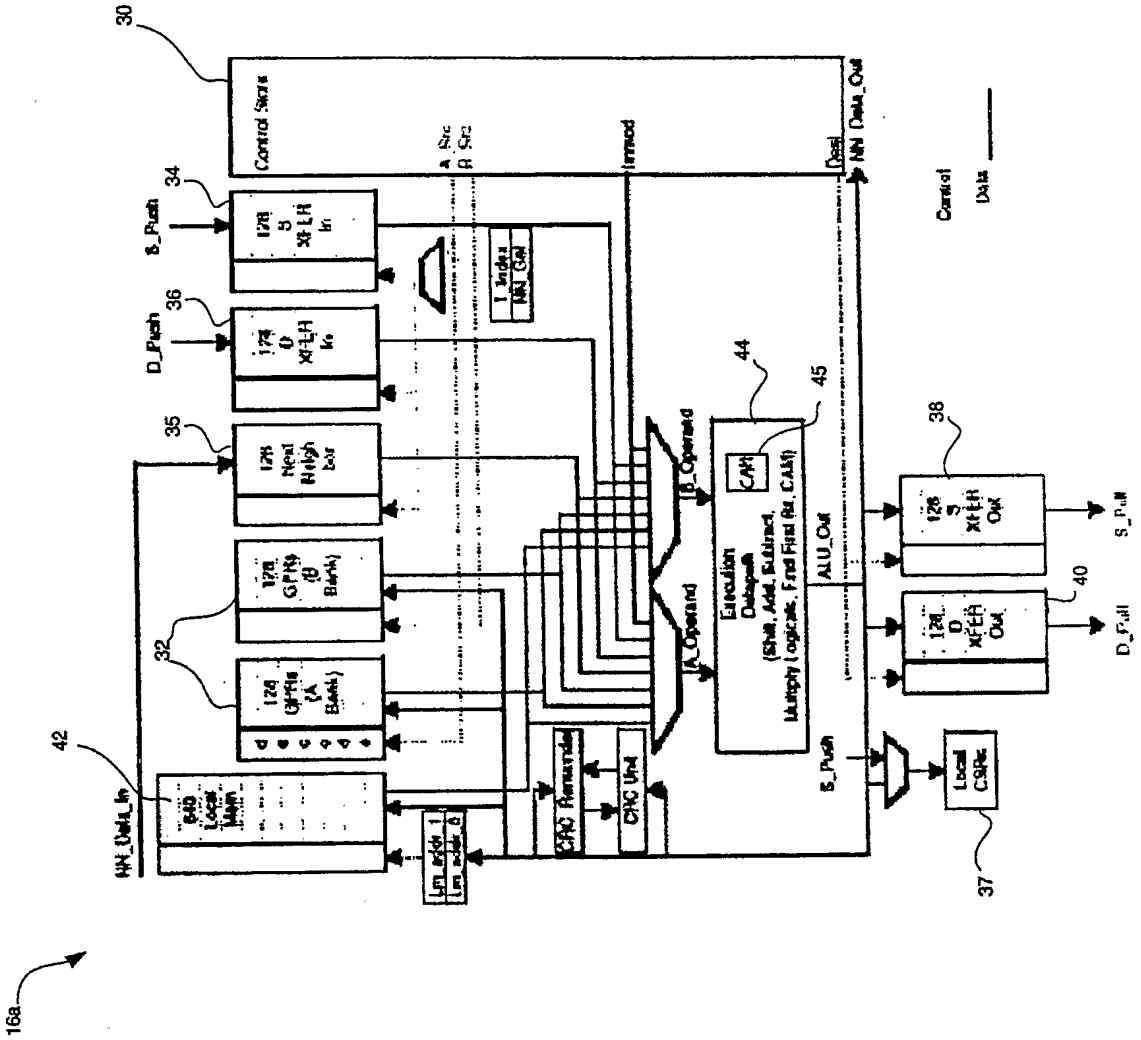

10
12
14
16
16a
16h
18a
18b
20
22
24a
24b
26a
26b
28a
28b
14a
14b
Processor
Core Processor
Programming Engines
DRAM Memory Controller
SRAM Memory Controller
DRAM
SRAM

FIG. 1

FIG. 2


30
Control Store
A Src
B Src
Immed
Dest
NN Data_Out
34
S_Push
128 S XFER In
36
D_Push
128 D XFER In
35
128 Next Neigh bor
32
128 GPRs (B Bank)
128 GPRs (A Bank)
42
NN_Data_In
64 Local Mem
Lm_addr_1
Lm_addr_0
CRC Remainder
CRC Unit
A Operand
B Operand
44
CAM
45
Execution Datapath (Shift, Add, Subtract, Multiply Logicals, Find First Bit, CAM)
ALU_Out
38
128 S XFER Out
40
128 D XFER Out
S_Push
D_Push
Local CSRs
37
Control
Data
16a

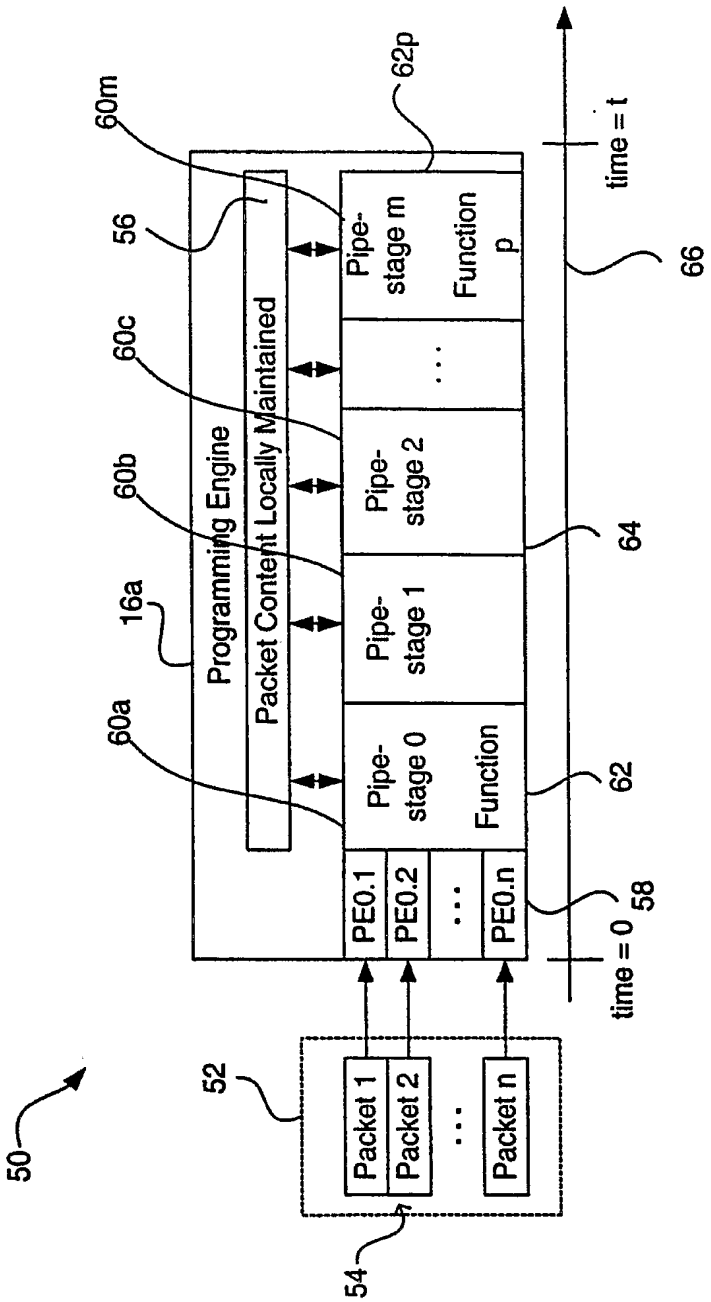

50
52
54
Packet 1
Packet 2
Packet n
PE0.1
PE0.2
PE0.n
58
time = 0
time = t
66
60m
56
60c
60b
16a
60a
Programming Engine
Packet Content Locally Maintained
Pipe-stage 0 Function
Pipe-stage 1
Pipe-stage 2
Pipe-stage m Function p
62
62p
64

FIG. 3

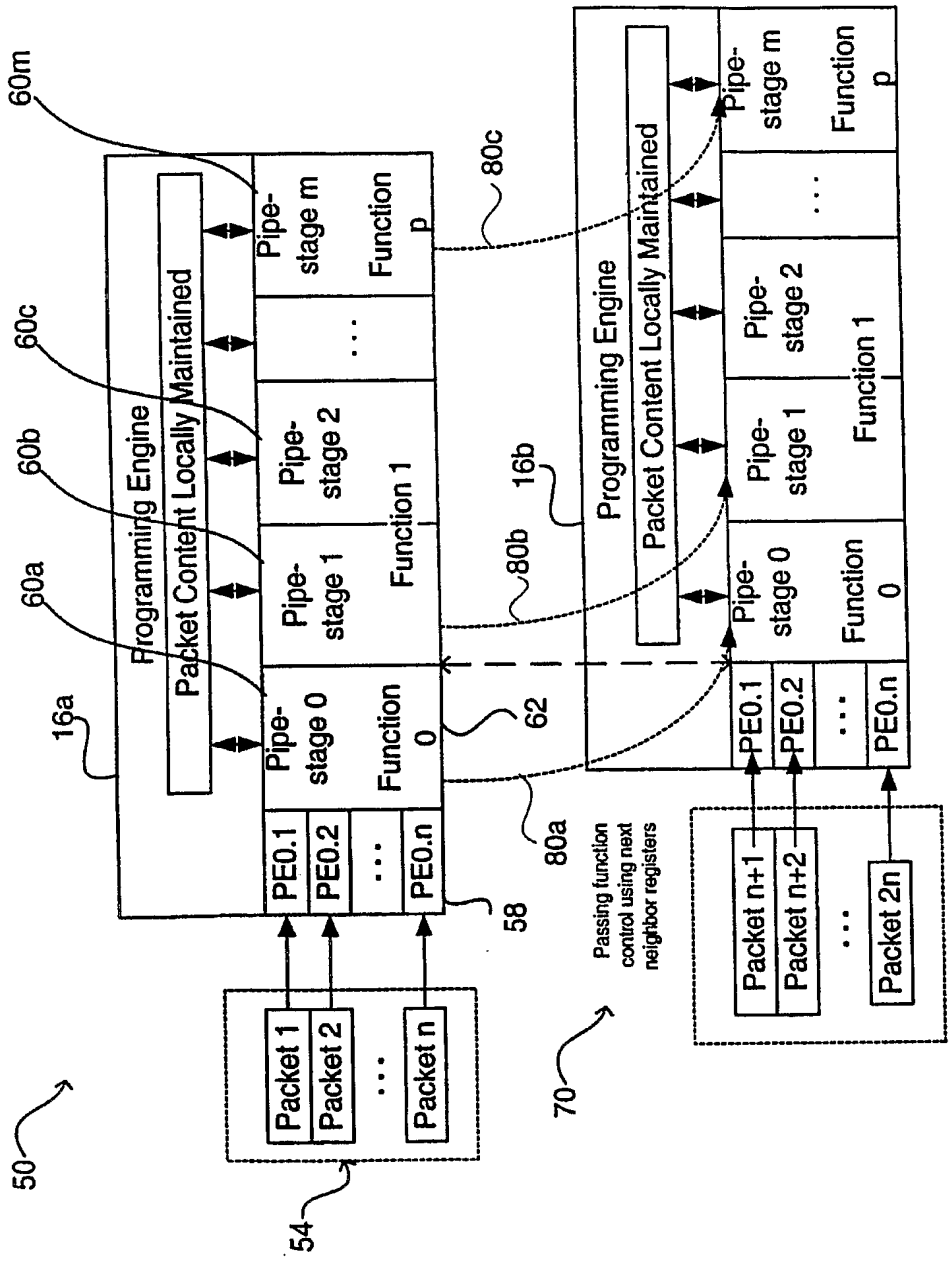

50
54
58
60a
60b
60c
60m
62
70
80a
80b
80c
16a
16b
Programming Engine
Packet Content Locally Maintained
Pipe-stage 0
Pipe-stage 1
Pipe-stage 2
Pipe-stage m
Function 0
Function 1
Function p
PE0.1
PE0.2
PE0.n
Packet 1
Packet 2
Packet n
Packet n+1
Packet n+2
Packet 2n
Passing function control using next neighbor registers

FIG. 4

FIG. 5

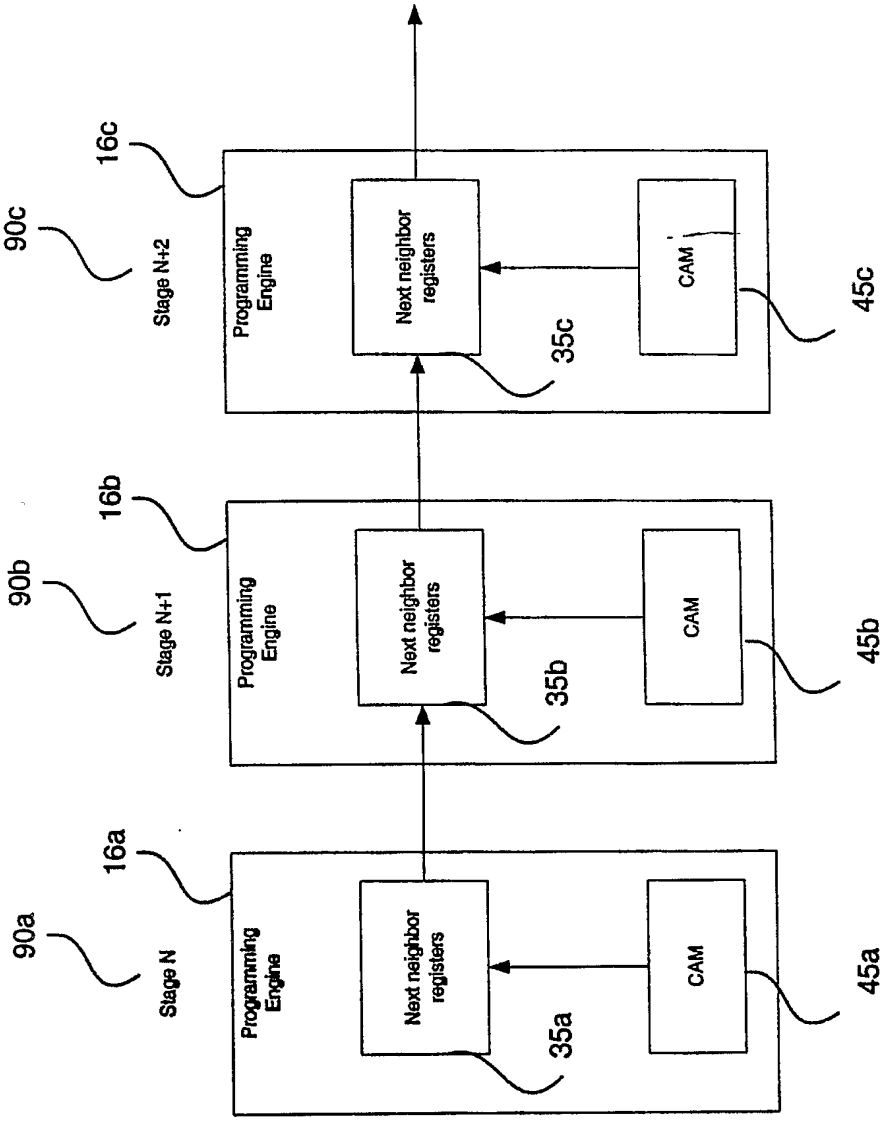

90a
Stage N
16a
Programming Engine
Next neighbor registers
35a
CAM
45a
90b
Stage N+1
16b
Programming Engine
Next neighbor registers
35b
CAM
45b
90c
Stage N+2
16c
Programming Engine
Next neighbor registers
35c
CAM
45c

FIG. 6

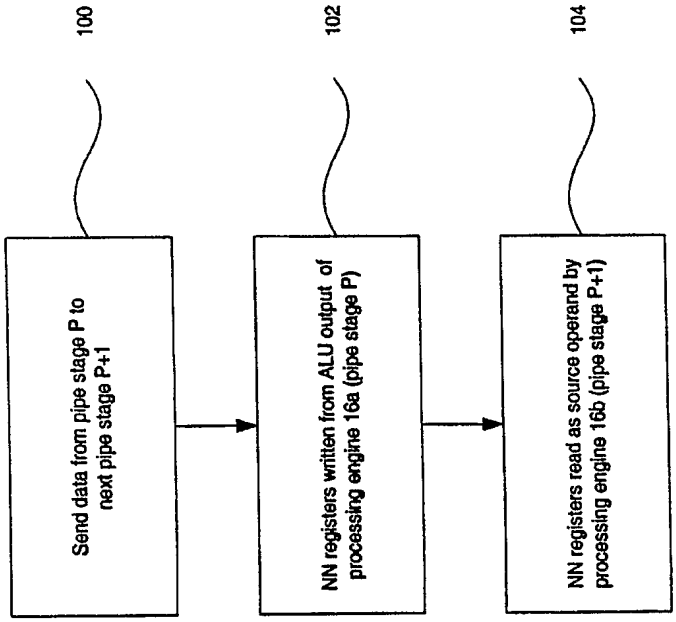
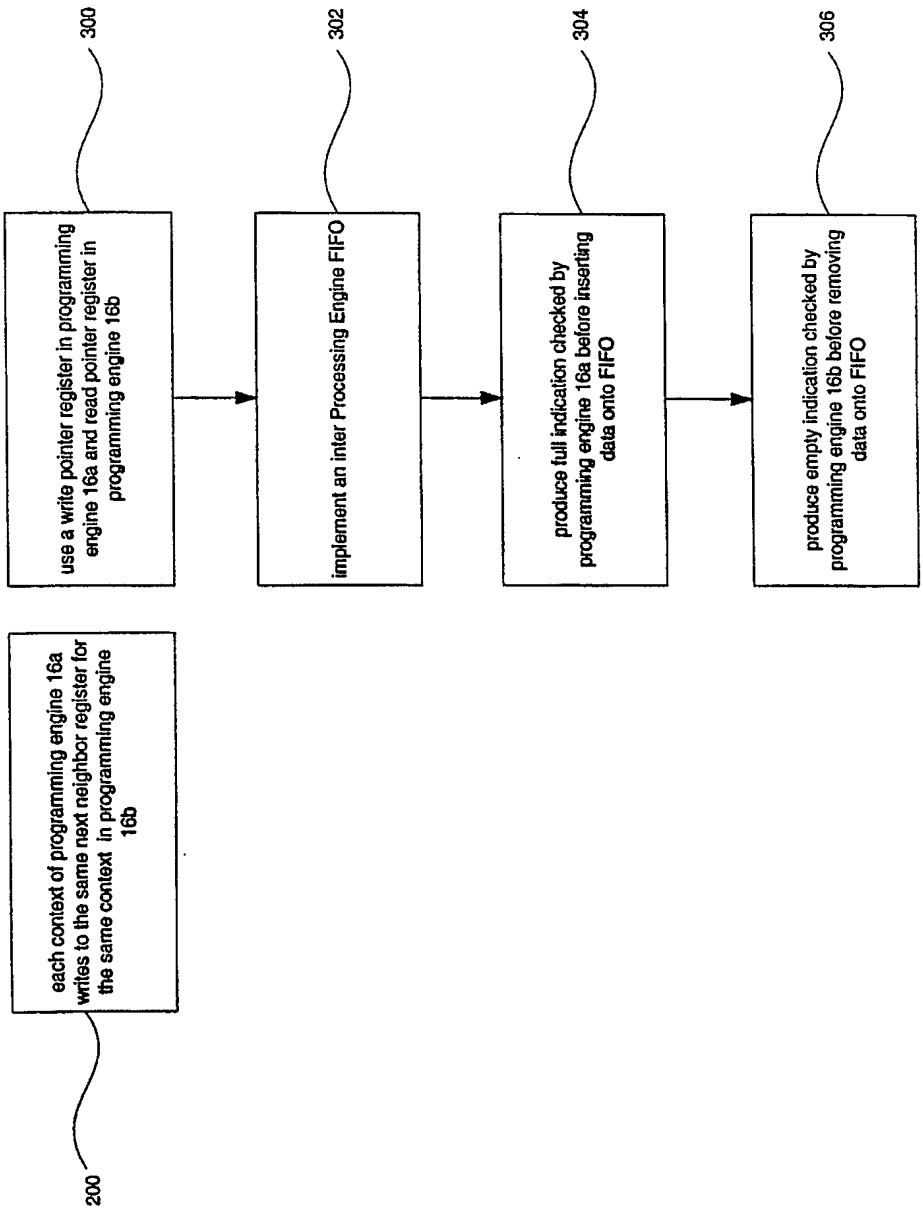

100
102
104
Send data from pipe stage P to next pipe stage P+1
NN registers written from ALU output of processing engine 16a (pipe stage P)
NN registers read as source operand by processing engine 16b (pipe stage P+1)

FIG. 7


200
each context of programming engine 16a writes to the same next neighbor register for the same context in programming engine 16b
300
use a write pointer register in programming engine 16a and read pointer register in programming engine 16b
302
implement an inter Processing Engine FIFO
304
produce full indication checked by programming engine 16a before inserting data onto FIFO
306
produce empty indication checked by programming engine 16b before removing data onto FIFO

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 03/09478

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/30 G06F9/38

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, INSPEC, IBM-TDB, COMPENDEX

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	WO 01 16718 A (BERNSTEIN DEBRA ;HOOPER DONALD F (US); INTEL CORP (US); WHEELER WI) 8 March 2001 (2001-03-08)	15-17, 21-24
Y	page 2, line 11 -page 3, line 17; figures 1,3-1,3-2 page 5, line 21 -page 6, line 12 page 15, line 17 -page 17, line 20	3-14, 17-20, 24-30

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
9 July 2003	17/07/2003

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31–70) 340–2040, Tx. 31 651 epo nl, Fax: (+31–70) 340–3016	Thibaudeau, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 03/09478

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
X	FARKAS K I ET AL: "The multicluster architecture: reducing cycle time through partitioning" PROCEEDINGS OF THE 30TH ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE. MICRO-30. RESEARCH TRIANGLE PARK, NC, DEC. 1 - 3, 1997, PROCEEDINGS OF THE ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITEC TURE, LOS ALAMITOS, CA: IEEE COMPUTER SOC, vol. 30TH CONF, 1 December 1997 (1997-12-01), pages 149-159, XP010261292 ISBN: 0-8186-7977-8	1,2,15, 16,21-23
Y	page 149, paragraph 1; figure 1	3-14, 17-20, 24-30
X	US 5 574 939 A (KECKLER STEPHEN W ET AL) 12 November 1996 (1996-11-12)	1,2,15, 16,21-23
Y	column 2, line 3 -column 3, line 13; figure 2	3-14, 17-20, 24-30
X	KECKLER S W ET AL: "Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor" COMPUTER ARCHITECTURE, 1998. PROCEEDINGS. THE 25TH ANNUAL INTERNATIONAL SYMPOSIUM ON BARCELONA, SPAIN 27 JUNE-1 JULY 1998, LOS ALAMITOS, CA, USA,IEEE COMPUT. SOC, US, 27 June 1998 (1998-06-27), pages 306-317, XP010291366 ISBN: 0-8186-8491-7	1,2,15, 16,21-23
Y	page 307, right-hand column, line 1 - line 43; figure 1 page 308, paragraph 2.2	3-14, 17-20, 24-30

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/US 03/09478

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 0116718 A	08-03-2001	AU 7061600 A	26-03-2001
		CA 2383384 A1	08-03-2001
		CN 1387641 T	25-12-2002
		EP 1221086 A1	10-07-2002
		WO 0116718 A1	08-03-2001
US 5574939 A	12-11-1996	WO 9427216 A1	24-11-1994